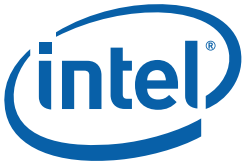

Intel® Thread Profiler

Guide to Sample Code

Document Number: 313104-001US

Revision: 3.0

World Wide Web: http://www.intel.com

Disclaimer and Legal Information

Revision History

Document Number	Revision Number	Description	Revision Date
		Initial releases.	2002-2005
313104-001 US	3.0	Updated for Intel® Thread Profiler 3.0 Product	July 2006

Contents

List of Figures

List of Tables

§

1 About this Document

This document presents examples to demonstrate how to detect typical multithreaded performance problems using the Intel® Thread Profiler. The goals of this document and associated code examples are to help you:

- Build code using the options required for the Intel® Thread Profiler.
- Recognize the type of performance issues that you can identify and analyze with the Intel® Thread Profiler.

To get the most from this document:

1. Review the sample code, typically installed in `C:\Program Files\Intel\VTune\tprofile\samples\`.
2. Build the sample code. Each sample includes a Microsoft* Visual Studio* workspace and project with required compiler options.
3. Generate data using the Intel® Thread Profiler. You can use the Wizard to create a project and Activity, as described in *Intel® Thread Profiler for Windows* Getting Started Guide*.
4. Explore the Activity results to understand the threading issues and solutions related to these code samples as described in this guide.

1.1 Intended Audience

This guide is intended for users of the Intel® Thread Profile who have read the *Intel® Thread Profiler Getting Started Guide*.

1.2 Using This Guide to Sample Code

This document describes the following code examples, according to types of threading:

Table 1 Document Organization

Sample Name	Threading Type
ompPrime	OpenMP*
pcOverlap, pcNoOverlap	Windows* Threading API

Sample Name	Threading Type
Hash, Sort	POSIX* Threads

For complete hardware and software requirements for using the Intel® Thread Profiler, consult the product release notes, available from, for example, **Start > Programs > Intel® Software Development Tools > Intel® Thread Profiler > Intel® Thread Profiler Release Notes**.

1.3 Conventions and Symbols

The following conventions are used in this document.

Table 2 Conventions and Symbols used in this Document

`This type style`	Indicates an element of syntax, reserved word, keyword, filename, computer output, or part of a program example. The text appears in lowercase unless uppercase is significant.
This type style	Indicates the exact characters you type as input. Also used to highlight the elements of a graphical user interface such as buttons and menu names.
This type style	Indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder.
[*items*]	Indicates that the items enclosed in brackets are optional.
{ *item* | *item* }	Indicates to select only one of the items listed between braces. A vertical bar (|) separates the items.
... (ellipses)	Indicates that you can repeat the preceding item.

1.4 Related Information

See the Intel® Thread Profiler *Documentation Index*, available from, for example **Start > Program Files > Intel® Software Developer Tools > Intel® Thread Profiler > Documentation and Support** for a complete listing of related documentation.

§

2 OmpPrime Sample Using OpenMP* Directives

This section examines three versions of a sample code that uses OpenMP* directives (pragmas) to split up the work of checking for prime numbers across multiple threads.

You will use Intel® Thread Profiler to:

- Identify performance issues in the code, such as load imbalance and lock contention.
- Understand different OpenMP* directives and their impact on performance.
- Learn how to improve the code's performance.

2.1 Build the ompPrime Samples

Before you begin using Intel® Thread Profiler, build `omPrime1`, `omPrime2`, and `omPrime3`, located by default in `tcheck\Samples\DataRaces`.

To compile:

1. Open the `OmPrime1.dsw` workspace in the Microsoft* Visual Studio Environment.
2. Click **Yes** to convert the workspace file into a solution (`.sln`) file.
3. Convert the project to use the Intel® C++ Project System in the Visual C++ .NET environment by right-clicking on it and selecting **Convert** or, in Visual Studio Environment 6, use the Intel® C++ Compiler Selection Tool.

The sample project is set up to use the `/Qopenmp` option. This flag is required when building your OpenMP* debug application for analysis using the Intel® Thread Profiler. Without it, the compiler ignores OpenMP* directives in your code.

2.2 Collect Data

Create a project and an Activity using Intel® Thread Profiler to generate data for each of the `Debug\` images you compiled: `omPrime1.exe`, `omPrime2.exe`, and `omPrime3.exe`.

You can use the **Intel® Thread Profiler Wizard** to create Activities using default values.

When prompted, make sure to:

- Set **Threading type** to **OpenMP* specific analysis (Supported Intel compilers only)**.
- Set the **Number of threads** to **2**.
- Click **OK** to accept the default value in the **MHz Overhead Estimates Parameters** dialog box.

Intel® Thread Profiler runs your Activity and presents the results in the **Timeline** and **Profile** views.

To add additional Activities to the same project, right-click the resulting project in the Tuning Browser and select **New Activity...** to launch the **New Activity** dialog box again.

TIP: For a review of how to create an Activity using Thread Profiler, see *Intel® Thread Profiler Getting Started Guide*.

TIP: If you need help completing the Intel® Thread Profiler Wizard, click the **Help** button on any page of the wizard for more details.

TIP: If Thread Profiler does not show results, select **Help > Search** and type "troubleshooting Thread Profiler" to search for relevant help topics. The help topic Troubleshooting Intel® Thread Profiler offers possible solutions.

TIP: To maximize the benefits of a threaded application, run this program on a multiprocessor platform, such as a system with an Intel® Pentium® 4 processor with Hyper-Threading Technology enabled or higher.

2.3 Analyze Results

The following sections explore the Activity results for the OpenMP* code samples included with Intel® Thread Profiler.

2.3.1 ompPrime1.c

This program finds all of the prime numbers in a range. By default, it finds the primes in the range one to 1,000,000. You can specify the start and end values of the range as the first two arguments for this program. You can also set an optional third argument which suppresses printing each prime number if set to zero. By default,

however, the program prints the total number of primes found, and the number of primes found in the forms 4n+1 and 4n+3.

The program determines if a number is a prime or not by dividing each potential prime number by all numbers less than or equal to its square root until a divisor is found. The larger the number being checked, the more the divisions performed. This is especially true for numbers that are found to be primes. Additionally, the program uses some mathematical shortcuts to only test odd numbers for primes, and to only divide by odd factors.

This program runs correctly, but is not optimized for threaded performance (speed). As shown in Figure 1, Summary view shows a load **Imbalance** (red) in the Activity results.

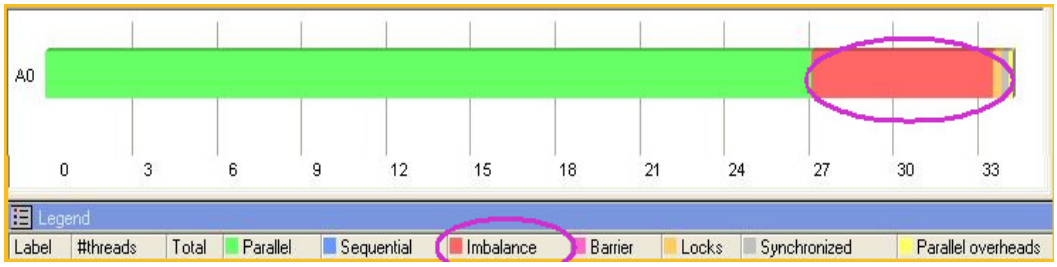

Figure 1: A load imbalance is indicated in Summary view by a red area in the bar graph.

You can click the **Threads** tab identify the exact thread that incurred the load imbalance as shown in Figure 2:

Figure 2: Threads view shows the location of the Imbalance on Thread T1.

When the program is run with two threads, one thread appears busy (checking a number to see if it is prime) in a parallel region, while the other thread stands idle for some amount of time.

Since the OpenMP* pragma:

```
#pragma omp parallel for private(factor, limit, prime)
```

in the sample code does not have a schedule clause, the application uses default scheduling. With a supported Intel compiler, threads are scheduled statically by default. Therefore the number of loop iterations is broken up evenly and assigned to each thread at the beginning of the loop. Since the number of divisions computed for each number varies, static scheduling can result in an unbalanced amount of work being given to the different threads.

2.3.2 ompPrime2Sched.c

The code in `ompPrime2Sched.c` is a slightly modified version of `ompPrime1.c`. It runs correctly, and runs faster. This version corrects the load imbalance by using the OpenMP* schedule clause with the OpenMP* `for` directive. Because the `for` loop's computation (workload) increases as its index increments, the default static scheduling does not work well. Instead, *dynamic scheduling* is used to account for varying amounts of work per iteration. This scheduling eliminates the load imbalance. But dynamic scheduling itself has more overhead associated with it than does static scheduling. Therefore a chunk size of `10` is added to reduce the overhead for dynamic scheduling as follows:

```
#pragma omp parallel for private(factor, limit, prime)\
schedule(dynamic,10)
```

You can drag the Activity result from the previous example for `ompPrime1.exe` from the **Tuning Browser** into Summary view to compare the performance of these two samples as shown in Figure 3.

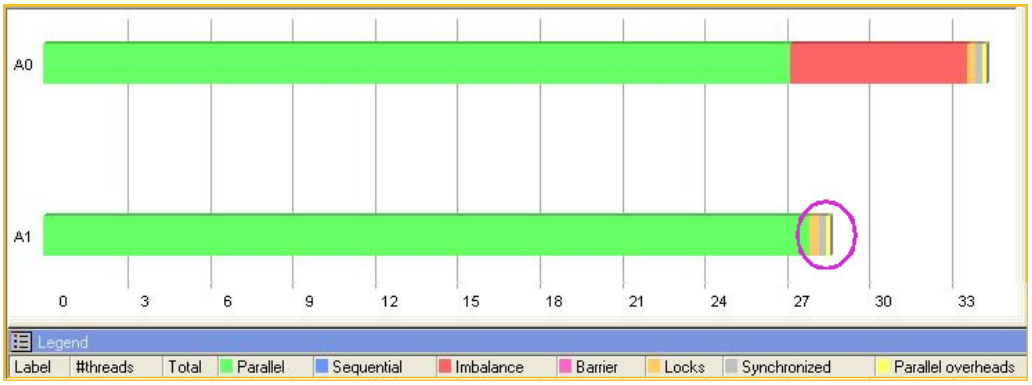

Figure 3: Summary view showing comparison of Activity results for the first and second examples. The Imbalance (red) has been eliminated in the second example, but a Lock issue (orange) remains.

The load **Imbalance** (red) present in **A0** in the first example is eliminated from this version of the program due to the use of dynamic scheduling, resulting in improved performance. However a major performance issue remains: **Lock** contention (orange) due to the OpenMP* critical directive.

The critical directive provides thread synchronization to prevent a data race for the variables `number_of_primes`, `number_of_41primes`, and `number_of_43primes`. Synchronization is a required overhead to enforce one-thread-at-a-time access to these variables, since they are all shared by all? the threads. These variables must be shared so that they can be updated (incremented) by each thread.

2.3.3 ompPrime3Opt.c

This is a slightly modified version of the `ompPrime2Sched.c`, that performs faster. This version eliminates the lock contention by replacing the OpenMP critical directive with a reduction clause on the OpenMP* `for` directive as follows:

```
#pragma omp parallel for private(factor, limit, prime) \
schedule(dynamic,10) \
reduction(+:number_of_primes,number_of_41primes,number_of_43primes)
```

As noted in the `ompPrime2Sched.c` sample discussion, the `number_of_primes`, `number_of_41primes`, and `number_of_43primes` counter variables must be synchronized because they are shared between threads and are incremented by each thread.

In this example, the reduction clause initializes to zero the private copies of the three counter variables for each thread and only updates the shared variables once at the end of the parallel for loop.

This program also runs correctly and has even better performance (speed) than the second sample. Notice that removing the lock overhead results in much better performance, as shown in Figure 4:

Figure 4: The third version of the code, corresponding to A2, has the best performance.

In the second version of the program, updating the shared variables, one thread at a time, on every iteration of the for loop caused lock contention. The reduction clause in this third version of the program causes the shared variables to be updated only once at the end of the loop, eliminating the lock contention and improving performance. Summary view now shows a **Parallel** time (green) for A2, without imbalance (red) or locks (orange), indicating well-tuned code with no threading performance issues.

3 pcOverlap and pcNoOverlap Samples Using Windows* Threading API

This code samples uses the Windows* threading API. It demonstrates how to detect threading performance bottlenecks using the Intel® Thread Profiler.

The source code for these examples is in, for example:

```
C:\Program Files\Intel\VTune\tprofile\samples\pcOverlap
C:\Program Files\Intel\VTune\tprofile\samples\pcNoOverlap
```

3.1 Build the PC Overlap Samples

To get the most out of threading, run this program on a multiprocessor platform, such as a system with an Intel® Pentium® 4 processor with Hyper-Threading Technology enabled or higher.

Invoke the Intel® C++ Compiler or the Microsoft* Compiler on the examples provided in the `<…>\tprofile\samples\pcOverlap` and `<…>\tprofile\samples\pcNoOverlap` directories, respectively, from the command line. For example, the following command compiles `ProducerConsumerNoOverlap.c` into the binary `ProducerConsumerNoOverlap.exe`:

```
icl /Zi /MD ProducerConsumerNoOverlap.c
```

3.2 Collect Data

To analyze performance, create and run a Thread Profiler Activity for each of the executable images you created. You can do so using the Intel® Thread Profiler Wizard.

CAUTION: In the wizard, make sure to set **Threading type** to **Windows* threads, POSIX* threads, and OpenMP* analysis**.

3.3 Analyze Results

The following sections describe the code and explore ways to use Thread Profiler to interpret performance data.

3.3.1 Task Parallel Code

A task parallel code achieves parallelism by decomposing the problem into different tasks and assigning one or more tasks to each thread. This sample uses a producer-consumer scenario as a task parallel example to illustrate the features of the Intel® Thread Profiler. The given problem is commonly used to demonstrate thread synchronization and includes two major tasks:

- producing data
- consuming data

Typically, one thread is assigned the task of producing data and copying them to a shared memory location. Another thread assumes the task of retrieving data from the shared location for consumption. These threads coordinate access to the shared location using locks or similar synchronization constructs to ensure that each access is atomic and that a produced piece of data is consumed before it is replaced by another piece of data.

3.3.2 Producer-Consumer Examples

The example includes two versions of a producer-consumer algorithm. It uses three threads:

- a producer thread
- a consumer thread
- a main thread that spawns the producer and consumer threads and then waits for them to terminate

The example uses two semaphores to synchronize access to the shared location in order to ensure a produced piece of data is not overwritten before it is consumed. The semaphores and their states are listed in Table 3.

Table 3: Semaphores used in the producer consumer examples.

Semaphore	State	Indicates
Empty	signaled	The shared location is empty and the producer can put a new piece of data in this location.
	unsignaled	The producer must wait until the consumer signals this semaphore.
Full	signaled	The shared location has an unconsumed piece of data.
	unsignalled	The consumer must wait until the producer signals the semaphore.

The producer waits for the empty semaphore to be signaled before writing a piece of data to the shared location and then signaling the full semaphore by invoking the function `ReleaseSemaphore`. It then starts the producing process again. The consumer waits until the full semaphore is signaled before reading the value from the shared location and then signals the empty semaphore once it consumes the value. Then, it restarts the consuming process. At the beginning of execution, the shared location is empty and the empty semaphore is signaled to allow the producer to proceed. Each time a thread acquires a signaled semaphore, that semaphore becomes unsignaled until it is signaled by a thread.

The two versions of the example differ in the overlap of execution between the producer and consumer threads:

- The **No Overlap** version, `ProducerConsumerNoOverlap.c` has no concurrent execution as only a single thread runs while the other one waits. Specifically, when the producer is generating and writing a datum to the shared location, the consumer stalls waiting for the full semaphore to be signaled by the producer. Conversely, when the consumer reads and consumes the piece of data, the producer stalls waiting for the empty semaphore to be signaled by the consumer.
- The **Overlap** version, `ProducerConsumerOverlap.c` enables concurrent execution of two threads by stalling the threads only when they access the shared location. In this example, the act of producing and consuming proceeds without stall. Specifically, the producer creates a piece of data and keeps it at a private location. Then it waits for the empty semaphore to be signaled before copying that piece of data to the shared location and signaling the full semaphore. The consumer reads the piece of data from the shared location after the full semaphore is signaled. It immediately signals the empty semaphore and then consumes the value.

NOTE: It is recommended that you run the same Activity at least two times and use the result of the latter run to avoid the overhead associated with the on-the-fly binary instrumentation associated with the first run using Thread Profiler. Instrumentation time contributes to the overall execution time of the program. Subsequent runs use the cached copies of the instrumented binaries and do not incur the cost of instrumentation.

3.3.3 Profile View

After running the Activity and getting results, the first view displayed is the **Profile View**. This view is initially grouped by *concurrency level*. The concurrency level is the number of active threads, that is, the number of threads that are not blocking or stalling.

TIP: If you change the grouping, you can switch back to group by Concurrency Level by clicking the  toolbar button.

In the Profile view, three bars indicate the amount of execution time for each of the three concurrency levels for the program `ProducerConsumerNoOverlap.exe` as shown in Figure 5.

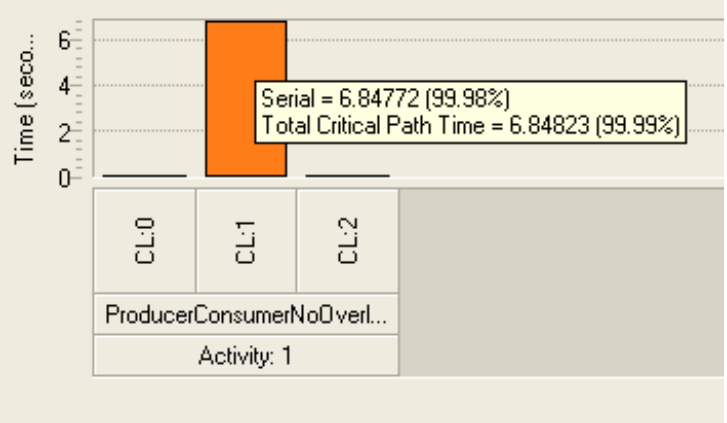

Figure 5: Profile view shows data by CL, concurrency level. Hover your mouse over a bar in the chart to show a tool tip.

For this example, during most of the execution, only a single thread was active. Although the example uses three threads, the execution was almost entirely **Serial** (orange). This view does not show which particular thread was active while it was on the critical path.

Since almost 100% of the runtime of this program was serial, it clearly needs optimizations to increase concurrency and effectively use the two processors on the system.

3.3.4 Grouping by Threads

To see which thread contributed to the serial time in this program, click the Thread button,  to group by threads. This grouping shows all the threads in a run. Each bar comprises of two portions that represent a thread:

- A portion with solid colors at the bottom of the bar that shows the time that the thread spent on the critical path.
- A semi-transparent portion on top (halo) that shows the time the thread spent off the critical path. The height of the bar (that combines the two portions) represents the lifetime of the thread.

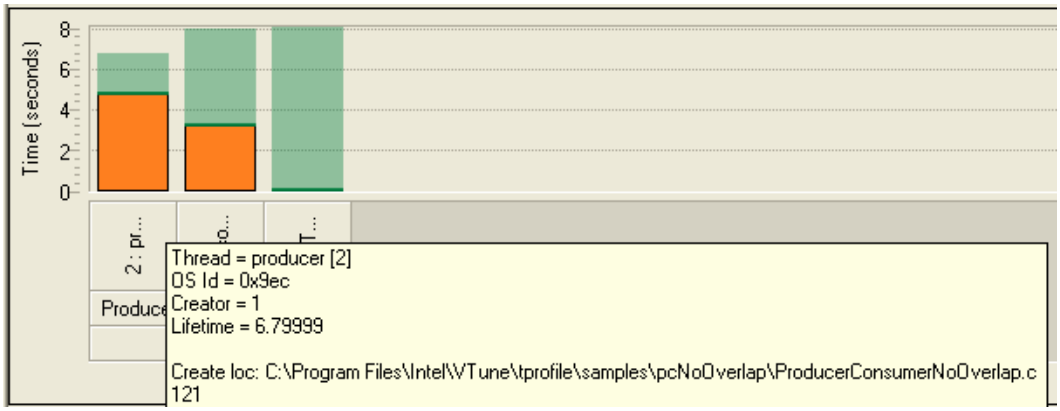

Figure 6: Profile view, with critical path time grouped by Thread, and Thread State checked in the Legend.

Figure 6 shows that the execution had three threads: Thread 1 on the right; Thread 2, and Thread 3. Thread 1 had the longest lifetime, but its time on the critical path was the shortest. Thread 2 had the shortest lifetime but the longest critical path time.

Hovering the mouse over the label below a bar displays the ID of the corresponding thread and its lifetime, the ID of a parent thread (creator) and the source code location at which the former thread was created by the parent thread.

Check the tooltip for Thread 2 to see that Thread 2 was created by Thread 1 and Thread 2's lifetime was 6.8 seconds. The source code at line 121 of file `ProducerConsumerNoOverlap.c` shows the following:

```
hThread[0] = CreateThread(0, 8*1024, producer, (void *)&p_id, 0, NULL);
```

Thread 2, therefore, is the producer thread.

Check the tooltip for Thread 3 to see that it is a consumer thread and Thread 1 is also its parent thread. The source code at line 122 of `ProducerConsumerNoOverlap.c` shows:

```
hThread[1] = CreateThread(0, 8*1024, consumer, (void *)&c_id, 0, NULL);
```

The tooltip for Thread 1 indicates that this thread is the top-level thread because its "Creator" is "(none)". In short, the execution had three threads: Thread 1 was the parent thread that created Thread 2 as a producer and Thread 3 as a consumer.

3.3.5 Grouping by Objects

Click the objects button,  to group by objects. This grouping shows the time associated with software objects that caused synchronization and threading delays as shown in Figure 7.

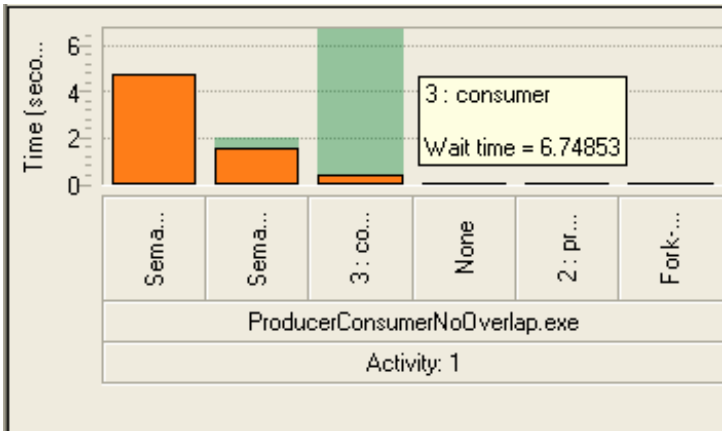

Figure 7: In this variation of Profile view, the critical path time is shown grouped by software object.

This view shows the following objects:

- two **Semaphore** objects
- one **None** object
- three **Fork-Join** objects

"None" includes the critical path time that is not attributable to any specific synchronization object. A Fork-Join object captures both thread creation time and thread joining time. A Semaphore object tracks the time it was used by a thread on the critical path to delay the next thread on the critical path. The color of the Fork-Join and Semaphore objects indicates that their critical path times were performed serially.

That is, the thread on the critical path was the only active thread. The heights of the halo bars are the lifetimes of the corresponding objects.

3.3.6 Creation Source View

To see the creation point of an object in the code, right-click a bar and select **Filter and Show Source Locations** as shown in

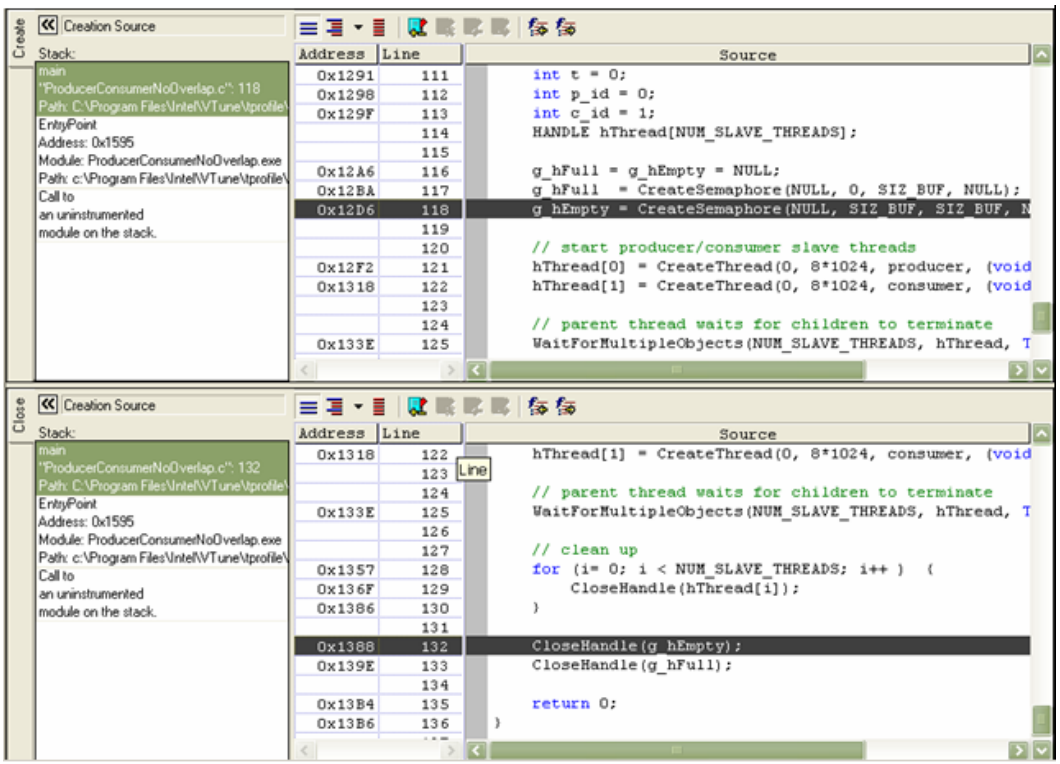

Figure 8: Creation Source view for the first Semaphore object in the example.

Figure 8 shows that the Sempahore 13 was created at line 118 of the source file `ProducerConsumerNoOverlap.c`. The code at that line indicates that the object is the Empty semaphore:

```
g_hEmpty = CreateSemaphore(NULL, SIZ_BUF, SIZ_BUF, NULL);
```

3.3.7 Grouping by Threads and Objects

In addition to the Objects view that shows the software objects that contributed to the critical path time, it is sometimes insightful to know the threads that used those software objects. To create this view:

1. In Profile view, click the Threads button, 
2. Click the Set Secondary grouping to object button,  on the right.

Now Thread is the first-level grouping and Objects is the second-level grouping as shown in Figure 9.

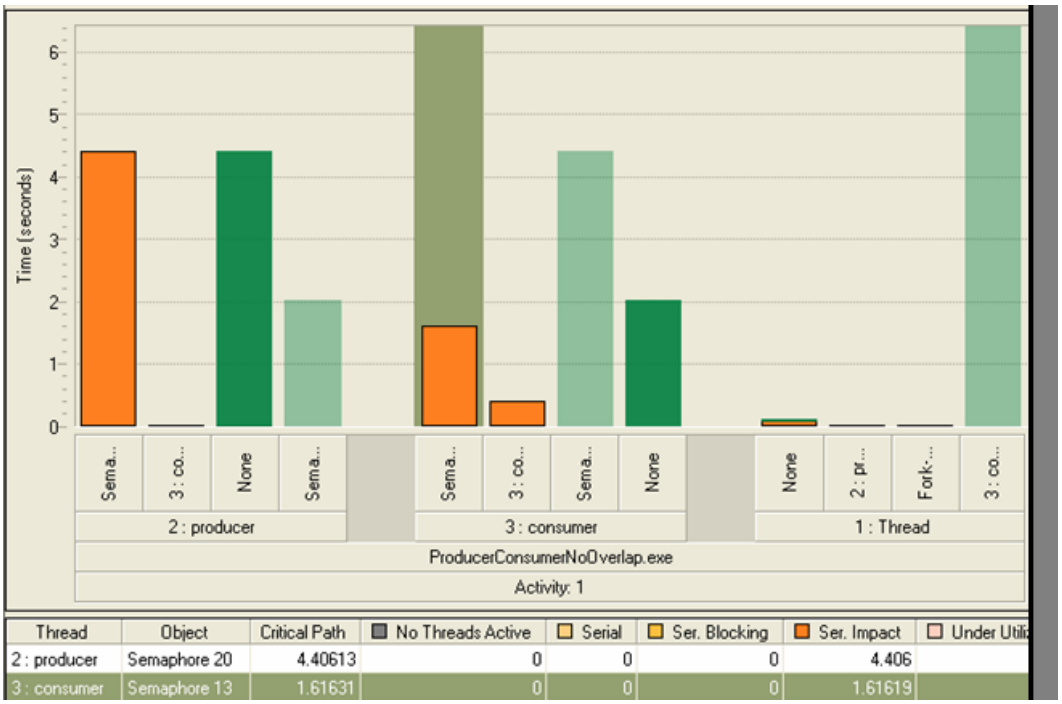

Figure 9: Profile view shown with Thread-Objects grouping. Selecting a bar populates the table below the legend with a row of statistics for the corresponding object.

The Thread-Objects grouping shown shows that Thread 1 spent only a short amount of time on the critical path. Thread 2's entire time on the critical path was attributed to a semaphore object. The object is Semaphore 20 and its Serial time is 4.4 seconds. Thread 3's critical path time spans over two objects: a Semaphore and a Fork-Join. Selecting its Semaphore object displays a row of statistic for this object in the table. This row shows that Semaphore 13 spent 1.62 seconds on the critical path.

What are Semaphore 20 and Semaphore 13? Placing the mouse over the Semaphore labels in the chart produces a tooltip. Based on the source locations listed in the tooltip, we know that Semaphore 22 is the full semaphore and Semaphore 8 is the empty semaphore.

Hovering the mouse pointer over the thread labels below the bars show the IDs and source location of the semaphores. Correlating the thread and semaphore information with the source code, we learn that Thread 2 was a producer and it used the Full semaphore to delay one or more threads. We also learn that Thread 3 was a consumer and it used Empty to delay one or more threads. The IDs of the delayed threads are not known until we filter the data by source location.

3.3.8 Filtering and Grouping by Source

Filtering and grouping by source location does the following:

- Filters out unselected bars.
- Applies the chosen grouping on the remaining bar(s).

To filter and group data by source location:

1. Select a bar or bars to keep.
2. Right-click and select **Filter and Group by >Source Stack**.

Figure 10 shows a single bar after the filter and grouping have been applied on Semaphore 13 of Thread 3.

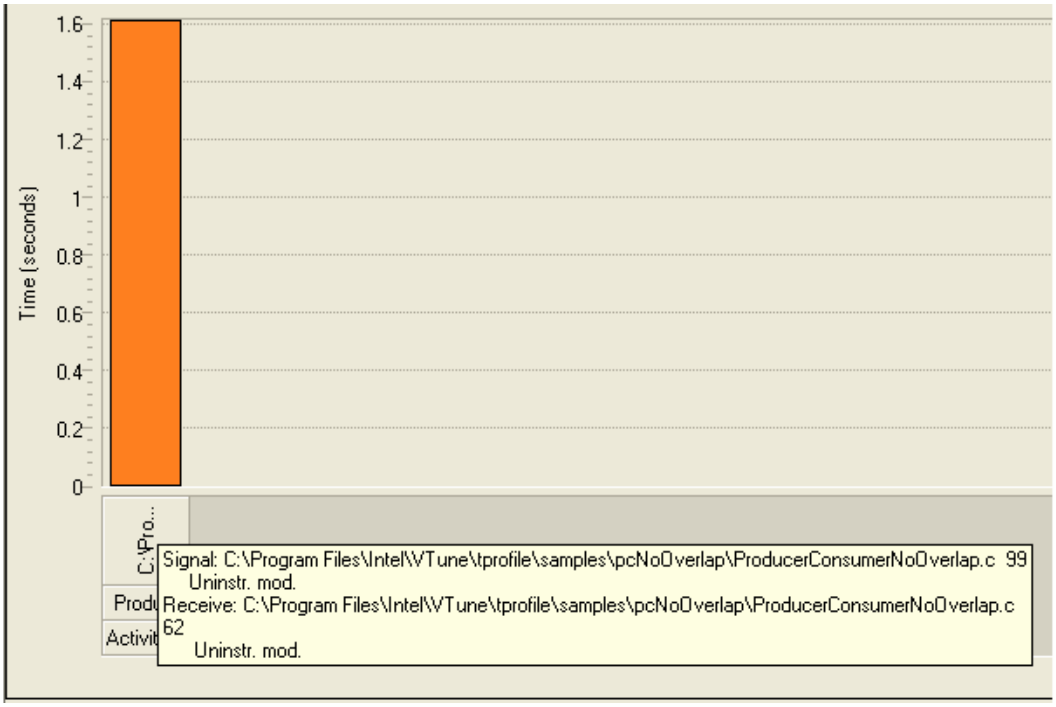

Figure 10: Result of a Filter and Group by Source

Single-click the bars to populate information about the source locations in the table below the chart. In order to see the source, double-click on a bar or select a bar, right click, and choose **Transition Source View**.

The Transition Source View shows two source windows. The information to the left of the source views shows the type of the source, the sender and the receiver of the signal, the associated object and the stack frames of the call site.

This call stack for the first source window has a single frame, which belongs to the consumer function in the file `ProducerConsumerNoOverlap.c`. Line 99 is the transition point of the critical path and is highlighted within the code segment of the consumer function. The code segment belongs to the consumer thread and the critical path transitioned out of this thread when it released the Empty semaphore.

The second source window shows the receive source which is the destination of the critical path transition. The next thread on the critical path is the producer. The critical path transitioned into this thread when the thread acquired the empty semaphore. This is the point where the producer was delayed until the consumer released the empty semaphore.

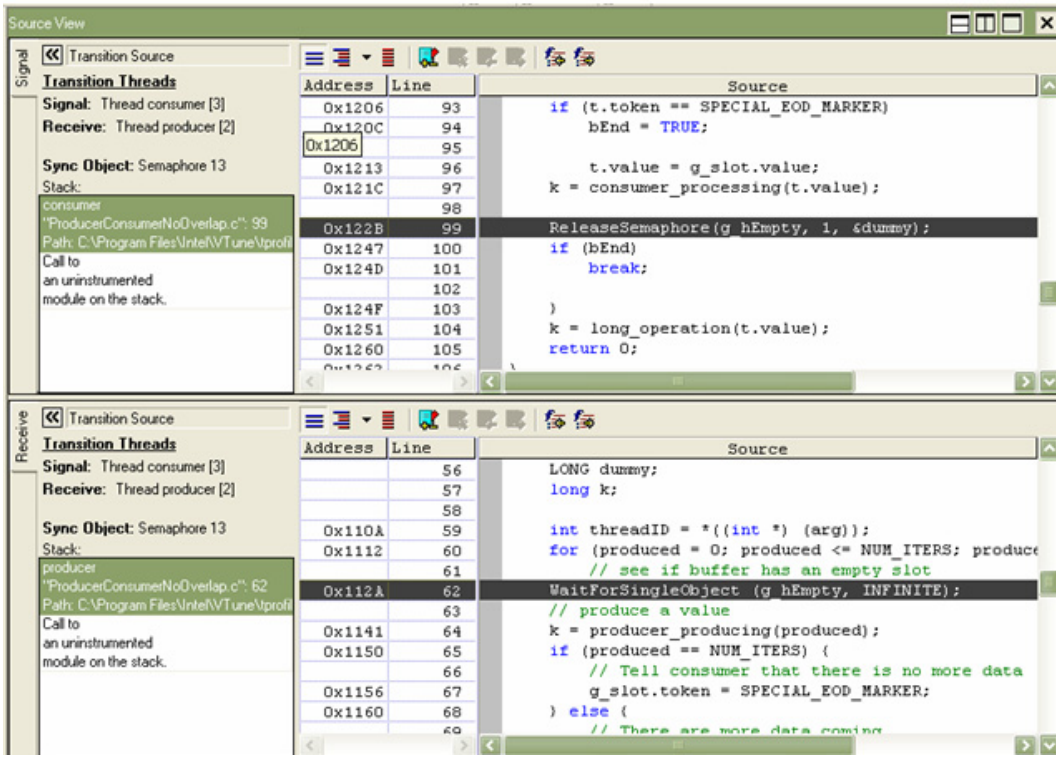

Figure 11: Transition Source View shows the signaling and receiving ends of the critical path transition.

You can optimize the example by moving the act of consuming in the consumer thread to a point after the release of the empty semaphore and similarly move the act of producing in the producer thread to a point before blocking on the empty semaphore. This optimization allows the consumer to signal the empty semaphore as early as possible and permits the producer to wait for the signal of the empty semaphore after it has already produced a piece of data. Essentially, the optimization enables the overlap of the access of the shared location with the act of producing/consuming. This optimization is used in the implementation of the version in `ProducerConsumerOverlap.c`.

3.4 Compare Results

To get the result for an example that overlaps the act of producing or consuming data with the act of accessing those data, run an Activity with the binary compiled from `ProducerConsumerOverlap.c`.

To compare the execution of `ProducerConsumerNoOverlap.c` with `ProducerConsumerOverlap.c`:

1. Double-click the Activity result of `ProducerConsumerNoOverlap.exe`.
2. Drag the Activity result of `ProducerConsumerOverlap.exe` into the chart area of the currently open consumer.

The result is shown in Figure 12.

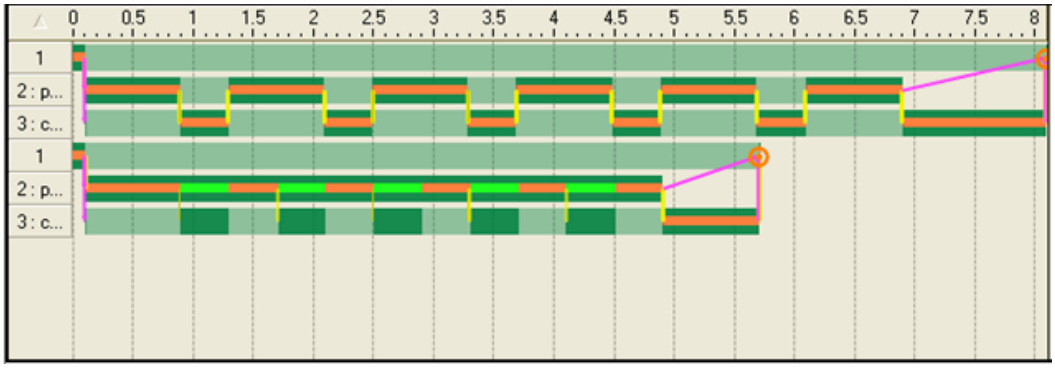

Figure 12: Timeline view comparing two Activity results.

Note that the execution of the overlap run (the bottom timeline) is shorter than that of the no-overlap run.

In Profile view, decomposition of the bars shows that the overlap one exhibits parallel execution for a duration of just over 2 seconds.

The Objects View shows that the impact of the Semaphore objects is much less in the overlap version. Most of the serial time has converted to parallel execution time associated with the consumer's fork-join object. The only semaphore with serial time is Semaphore 14, which is the Full semaphore. You can confirm this by selecting the bar of this semaphore, right-click and select **Filter and Group by > Object** to view the creation of the following semaphore:

```
g_hFull = CreateSemaphore(NULL, 0, SIZ_BUF, NULL);
```

The serial time attributed to the Empty semaphore in the no-overlap version disappeared in the overlap version because the consumer thread no longer delays the producer thread. Recall that we have optimized the code to have the consumer signal the Empty semaphore after it accesses the shared location and before it processes data. The optimization also allows the producer to produce a piece of data before accessing the shared location.

If you filter out the rest of the bars except for the one corresponding to the Full semaphore and then group by Source, you can see the call sites to which this serial time is attributed. In this case there is just one: when the producer thread delayed the consumer thread which was waiting for the Full semaphore. The only way to

reduce this serial time is to make the producer produce data faster by improving the algorithm which comes up with the data value.

TIP: For more information about other types of performance issues see the product online help. **Help** also describes additional features not explored in this example.

4 Hash Samples Using Windows* API or POSIX* Threads

This section uses code samples to demonstrate how to detect threading performance issues using the Intel® Thread Profiler. You can choose from versions of the sample code that uses Windows* API or Pthreads.

NOTE: To maximize the benefits of a threaded application, run this program on a multiprocessor platform, such as a system with an Intel® Pentium® 4 processor with Hyper-Threading Technology enabled or higher.

The hash sample code is installed by default in the following folders:

- Windows* API version: `C:\Program Files\Intel\VTune\tprofile\samples\hash` folder.
- Pthreads version: sample code is installed with Intel® Thread Profiler for Linux* product, typically in `/opt/intel/itt/tprofile/samples`. See `/opt/intel/itt/tprofile/samples/Readme.txt` for compile instructions on Linux systems.

4.1 Build Hash Code

Invoke the Intel® C++ Compiler or the Microsoft* Compiler on the sample code files examples provided in `C:\Program Files\Intel\VTune\tprofile\samples\` from the command line.

For example, the command below compiles `sort.c` into the binary `sort.exe`:

```
icl /Zi /MD sort.c
```

4.2 Collect Data

In the Intel® Thread Profiler, use the wizard to create an Activity for one of the executable images you created using the sample code and run the Activity to collect results.

CAUTION: In the wizard, make sure to set **Threading type** to **Windows* threads, POSIX* threads, and OpenMP* analysis**.

4.3 Analyze Results

The following sections describe the Windows* API threaded code samples and explore the Activity results, and how to best use the Intel® Thread Profiler's features to interpret performance data.

4.3.1 About the Hash Example

The "hash" example code generates a number of threads to load a hash table. The hash table is partitioned into sub-tables, each with its own mutex lock. When a thread wants to write to a sub-table, it first acquires the lock.

You can adjust the number of worker threads, `N_THREADS`, and the amount of time spent doing busy work, `BUSY_TIME`, to produce different results.

To set these parameters, use the following command line arguments:

```
hash.exe [-n number_of_threads] [-b busy_time_integer ]
```

4.4 Profile View

Run the hash example with the default number of threads, six. The first view you see when opening an Activity result is the Profile View and Timeline view as shown in Figure 16. In Profile view, a bar represents the total execution time of the program `hash.exe` grouped by the concurrency level. For this example, during most of the execution, there were more active threads than the number of available processors, and most of the rest of the time had only a single thread operating (serial time) as shown in Figure 13.

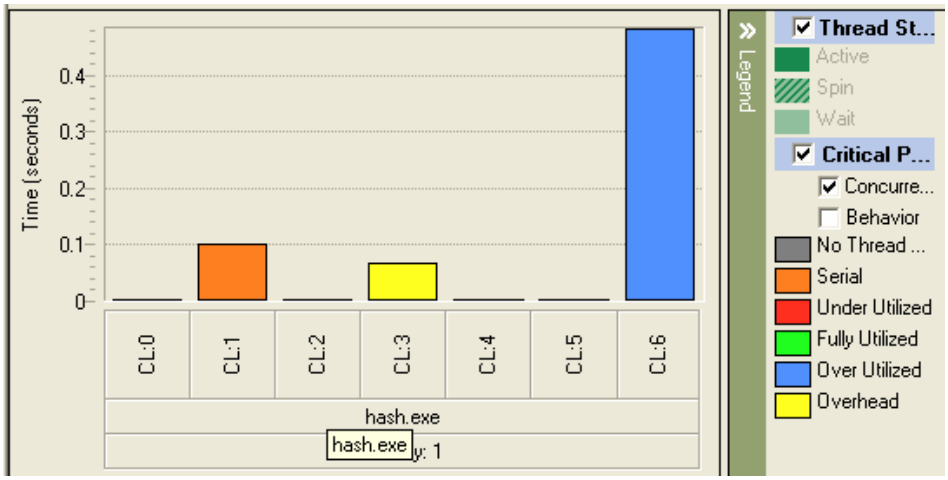

Figure 13: Initial results for the hash example show room for improving performance.

Use the **Legend** on the right-side of the view to interpret the colors:

- Blue bars indicate **Over Utilized & Impact** time. During this time the number of active threads is greater than the number of processors.
- Orange indicates **Serial** time. During this time, only one thread is active.
- Yellow indicates **Overhead** time caused by excessive concurrency.

This program could use some optimizations to reduce excessive concurrency. Too much concurrency creates unnecessary overhead as indicated by the yellow bar. If you rerun the Activity but change the command line parameters to use two threads so that "-n 2". You will see a very different Profile view, as shown in Figure 1.

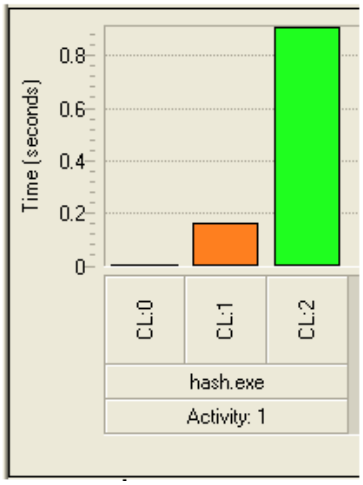

Figure 14: the hash example using only two worker threads.

After changing the command line arguments to only use two worker threads, note that almost all of the time is **Fully Utilized & parallel** (green) execution. If you compare both the runs side by side by dragging the second Activity result on top of the original one, **Timeline** view shows that the second example does indeed run in less time, as shown in Figure 15. This is primarily because the second run does not suffer from the overhead due to excessive synchronization.

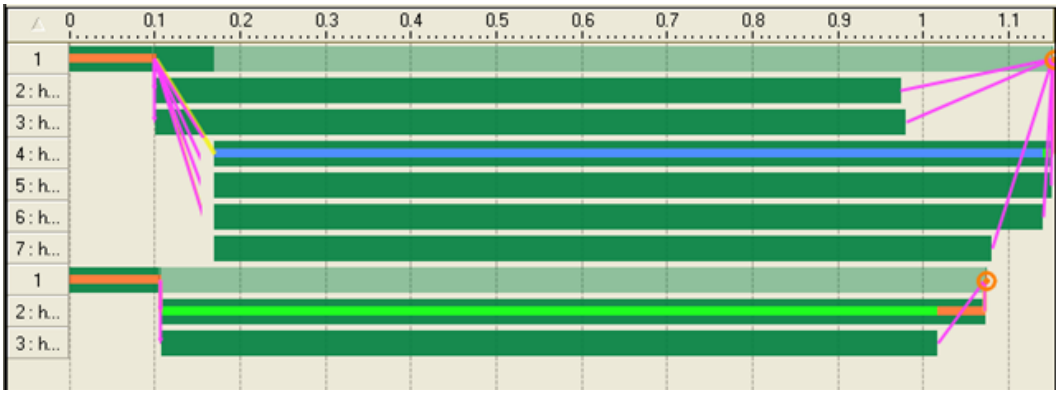

Figure 15: A side-by-side comparison of the two runs of the hash example with different *n* values.

If the number of threads is set to a very large number such as 100, the difference between the run with two threads is even more pronounced.

5 Additional Sample: Sort

An additional code example, Sort, is provided for you to explore on your own in the `\samples\sort` folder.

You are also encouraged to create an Activity, analyze the performance of the sort sample code, and explore results using the techniques described for the hash. In this example, threads created early wait for later threads to complete their tasks before proceeding. The thread sets an event object for each thread, initially to the nonsignaled state. When a thread completes its work, it sets its Event to the signaled state. All waiting threads can detect that signaled state via `WaitForSingleObject()`.

NOTE: The Linux* version of the sample code uses POSIX* (Pthreads*) conditional variables to perform the same tasks as events in the Windows* API version without changing program logic. Since both versions of the code use the same algorithm you can apply the same analysis for both versions. However, due to the OS differences, different techniques are used to implement the algorithm. As a result, you should see similar graphical representations of results in Intel® Thread Profiler, but may see conditional variables as a synchronization object instead of events.

NOTE: It is recommended that you run the same Activity at least two times and use the result of the latter run to avoid the overhead associated with the on-the-fly binary instrumentation. When a program is monitored by Intel® Thread Profiler for the first time, Intel® Thread Profiler instruments the program binary and associated user and system DLLs on-the-fly and caches the instrumented copies for subsequent runs. Instrumentation time contributes to the overall execution time of the program. Subsequent runs use the cached copies of the instrumented binaries and do not incur the cost of instrumentation.